

**Calcolatori Elettronici T  
Ingegneria Informatica**

# **03 Linguaggio macchina**



# Instruction Set Architecture

- L'insieme delle istruzioni e dei registri di una CPU costituiscono l'*Instruction Set Architecture* (ISA)
- Mediante l'ISA è possibile accedere alle risorse interne (e.g., registri) ed esterne (e.g., memoria)
- Tipicamente le istruzioni in linguaggio macchina sono generate da un compilatore
- Più raramente, come in questo corso, scritte dai programmatori
- Purtroppo, (quasi) ogni CPU possiede un proprio ISA
- A proposito di ISA, esistono due linee di pensiero:
  - **RISC**: insieme ridotto di istruzioni semplici -> molti registri interni (DLX, ARM, RISC-V, etc)
  - **CISC**: insieme ampio di istruzioni complesse -> pochi registri (Intel X86)

# Requisiti di un linguaggio macchina/ISA

Oltre alla possibilità di poter risolvere un qualsiasi problema\*, un requisito fondamentale di un linguaggio macchina/ISA è quello di minimizzare il tempo di esecuzione del codice\*

- Se  $CPI_{medio}$  è il numero medio di clock per l'esecuzione di una istruzione, l'obiettivo è quello di minimizzare

$$CPU_{Time} = N_{istruzioni} * CPI_{medio} * T_{CK}$$

- Lo stesso problema, può essere quindi risolto con  $CPU_{Time}$  diversi in base a:
  - $N_{istruzioni}$  (RISC, richiedono in genere più istruzioni)
  - $CPI_{medio}$  (RISC, tipicamente istruzioni più veloci)
  - $T_{CK}$  (Reti logiche semplici potenzialmente più veloci)

# Istruzioni e risorse interne a una CPU

Le istruzioni eseguibili da una CPU, codificate in binario, sono in genere molto più semplici delle istruzioni che utilizzate nei linguaggi ad alto livello.

Tipiche operazioni sono:

- somme, sottrazioni, divisioni, moltiplicazioni, etc
- letture e scritture in memoria e periferiche
- confronto tra operandi ("A>B?")
- salti condizionati ("salta se") e incondizionati ("salta")
- . . .

E' possibile, mediante le istruzioni, accedere a risorse interne della CPU come registri architetturali e talvolta a registri di stato (e.g., A era maggiore di B?)

Le risorse che sono accessibili alle istruzioni in linguaggio macchina sono definite dai progettisti dalla CPU

Tuttavia, non tutti i registri interni sono accessibili al programmatore (senza che questo rappresenti un problema)

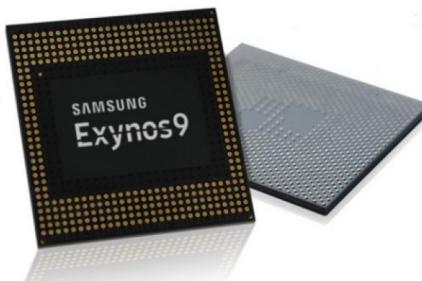
# Registri di una CPU

- Ogni CPU possiede un certo numero di registri accessibili al programmatore
- Il numero e la dimensione dei registri dipendono dall'ISA (e questo ha impatto sulla rete logica risultante)
- Ovviamente, avere *multi* registri *general purpose* (GP) è vantaggioso (meno accessi alla lenta memoria)
- Avere istruzioni che possono usare tutti, o quasi, i registri GP senza vincoli è vantaggioso
- Lo stesso ISA può essere realizzato con reti logiche completamente differenti (e.g., Intel e AMD)
- Queste reti hanno in genere prestazioni diverse (diverso  $CPU_{Time}$  sebbene abbiano stesso  $N_{istruzioni}$ )
- Non sarebbe stato meglio avere un solo ISA?



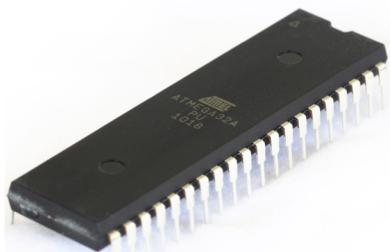
Desktop

IA: Intel X86 (CISC)



Smartphone  
e Tablet

IA: ARM (RISC)



Arduino Uno

ATMEL (RISC 8 bit)

## RISC-V (1/2)

- Il progetto RISC-V mira proprio a questo: creare un ISA unico e open source
- Ovviamente l'obiettivo non è quello di uniformare le reti logiche che implementano l'ISA
- L'ISA base del progetto RISC-V è molto simile a quella del DLX che studieremo e progetteremo in questo corso
- Esistono specifiche per estendere l'ISA base al fine di contemplare particolari funzionalità (*floating-point, Single Instruction Multiple Data (SIMD), 32/64/128 bit, etc*)



RISC-V: The Free and Open RISC Instruction Set Architecture

# RISC-V (2/2)



## Foundation Members

Clip slide

### Platinum Founding Sponsors



Western  
Digital®

D R A P E R

Microsemi



Hewlett Packard  
Labs

ORACLE

Rambus  
Google

SiFive

Mellanox  
TECHNOLOGIES

### Gold & Silver Founding Sponsors



ROA  
LOGIC



Rumble  
Development



LATTICE  
SEMICONDUCTOR



BAE SYSTEMS

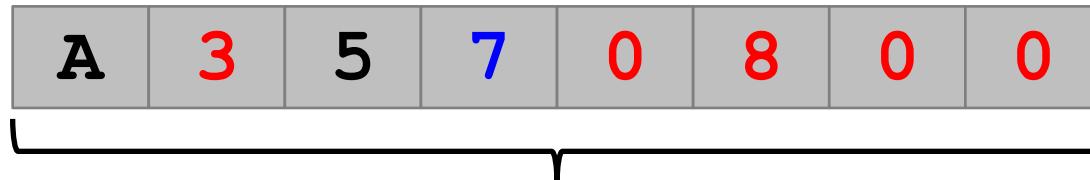
Technolution



# Codifica binaria delle istruzioni

- Le istruzioni per essere eseguite dalla rete logica CPU debbono essere codificate in binario secondo un formato noto e documentato dal produttore (*datasheet*)
- La codifica binaria deve contenere tutte le informazioni necessarie all'**Unità di Controllo** per poter eseguire l'istruzione
- Esistono CPU con codifica delle istruzioni a lunghezza:
  - costante** (e.g., 32 bit caso RISC DLX e molti altri)
  - variabile da istruzione a istruzione** (Intel X86)

Esempio: LB R7,0800(R3) - "Leggi un BYTE (8 bit) all'indirizzo R3 + 0800h e trasferisci nel registro R7"



Ipotetica codifica dell'istruzione con 32 bit. I bit non utilizzati per codificare R3, R7 e 0800 rappresentano il codice operativo (op code)

# Linguaggio assembler

- La codifica delle istruzioni in linguaggio macchina è poco intuitiva per gli esseri umani
- Nel linguaggio\* **assembler** si codificano le informazioni in un modo (un po') più intuitivo

Macchina → Assembler

014FA27Dh → ADD R1,R2,R3 ; R1=R2+R3

Escludendo la caratteristica appena evidenziata, un altro significativo vantaggio è quello di poter definire delle **label** utili (spesso) nei salti

LOOP: SUB R1,R1,R3

... . . . . .

BNEZ (R1),LOOP ; salta a LOOP se R1!=0

# Notazione per la costruzione di configurazioni binarie

Consente di esprimere efficacemente configurazioni binarie:

- Traslazione logica a sinistra di n bit:  $\ll n$   
(inserendo "0" a destra)
- Traslazione logica a destra di n bit:  $\gg n$   
(inserendo "0" a sinistra)
- Concatenazione di due campi:  $\# \#$
- Ripetizione n volte di x:  $(x)^n$
- Ennesimo bit di una conf. binaria x:  $x_n$   
(il pedice seleziona un bit)
- Selezione di un campo in una stringa di bit x:  $x_{n..m}$   
(un range in pedice seleziona il campo)
- Data la configurazione binaria di 8 bit C = 01101100<sub>2</sub>:
  - C  $\ll 2$  : 10110000<sub>2</sub>
  - C<sub>3..0</sub>  $\# \#$  1111 : 1100|1111<sub>2</sub>
  - (C<sub>3..0</sub>)<sup>2</sup> : 11001100<sub>2</sub>
  - (C<sub>6</sub>)<sup>4</sup>  $\# \#$  C $\gg 4$  : 1111|00000110<sub>2</sub>

# Altra notazione

- **Trasferimento di un dato:**  $\leftarrow$

Il numero di bit trasferiti è dato dalla dimensione del campo destinazione; la lunghezza del campo va specificata secondo la notazione seguente tutte le volte che non è altrimenti evidente

- **Trasferimento di un dato di n bit:**  $\leftarrow_n$

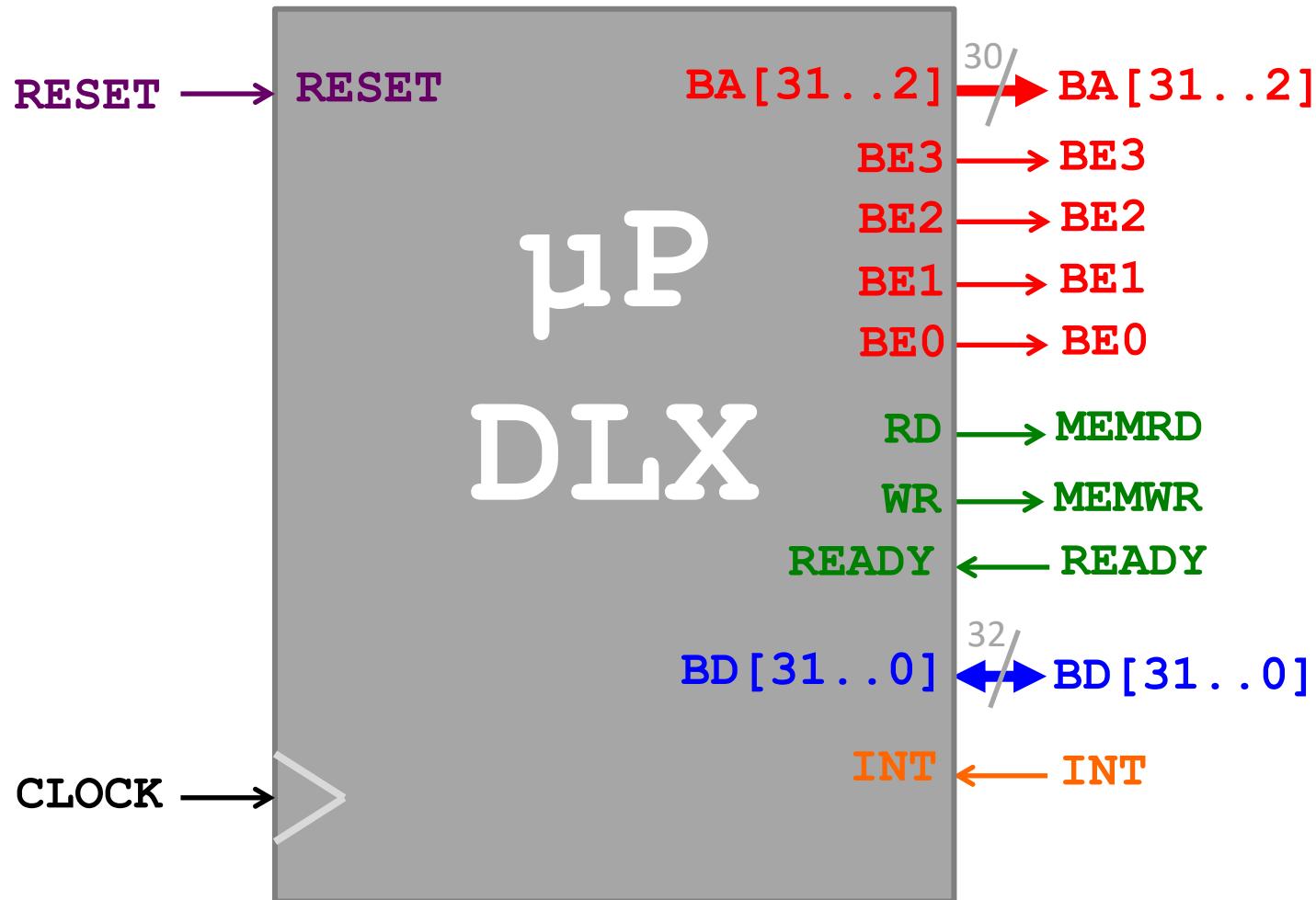
Questa notazione si usa per trasferire un campo di n bit, tutte le volte che il numero di bit da trasferire non è evidente senza la relativa indicazione esplicita

- **Contenuto di celle di memoria adiacenti a partire dall'indirizzo x:**  $M[x]$

## Esempio:

$R1 \leftarrow_{32} M[x]$  indica il trasferimento dalla memoria verso il registro R1 dei 4 byte:  $M[x]$ ,  $M[x+1]$ ,  $M[x+2]$ ,  $M[x+3]$

# Segnali del processore DLX

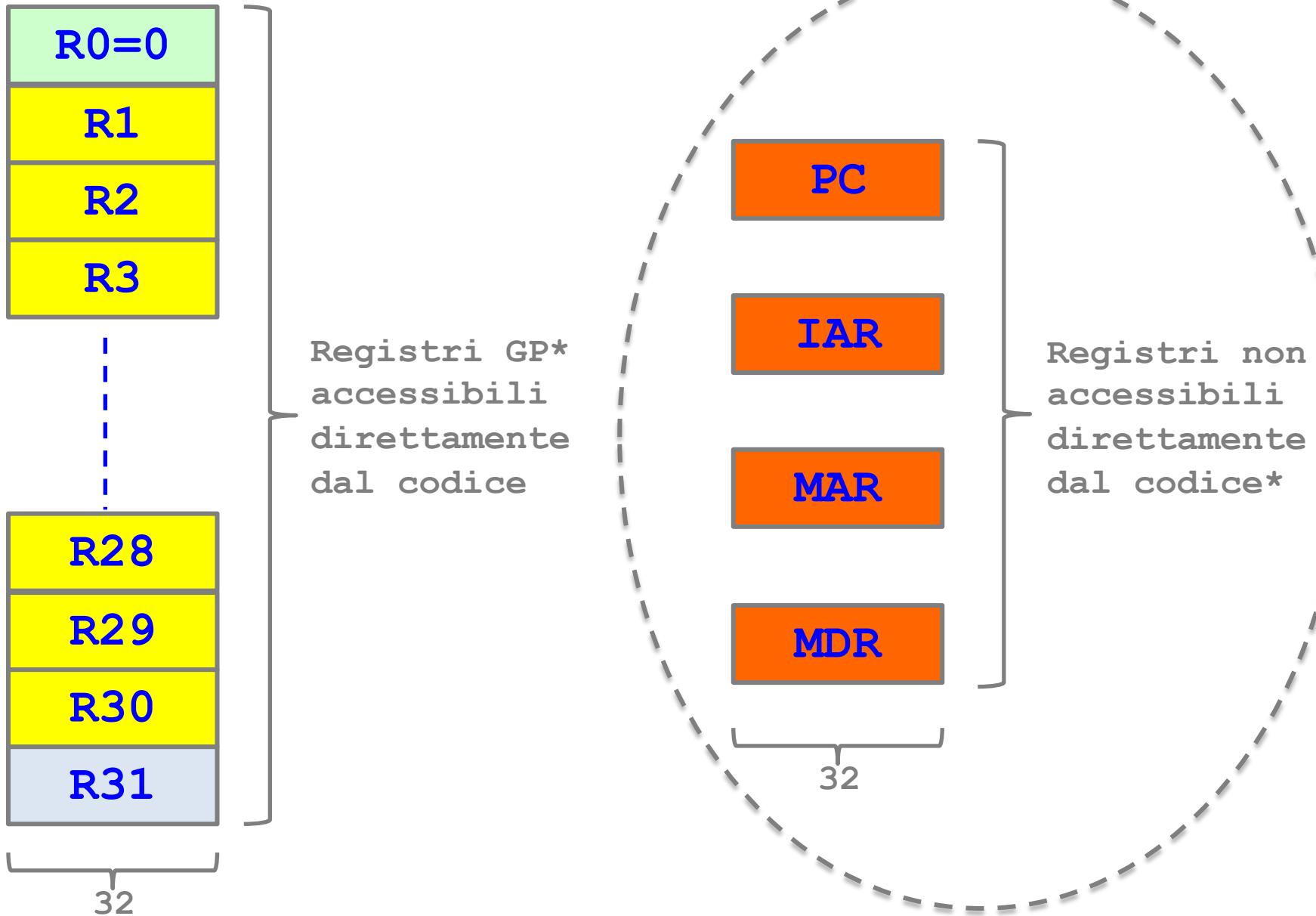


Il segnale di **RESET** è asserito, **all'avvio**, da una rete esterna. Anche i segnali di **READY**, **INT** sono generati da reti esterne ma utilizzati durante il normale funzionamento.

# Caratteristiche dell' ISA DLX (integer)

- Unico spazio di indirizzamento di 4G
- 32 registri da 32 bit GP (R0,...,R31, con R0=0)
- Istruzioni di lunghezza costante, 32 bit allineate
- Campi delle istruzioni di dimensioni/posizioni fisse
- 3 formati di istruzione: I,R,J
- Non ci sono istruzioni per gestire lo stack
- Per istruzioni che prevedono un indirizzo di ritorno (JAL/JALR), esso è salvato in R31
- Non esiste un registro di FLAG settato dalle istruzioni ALU. Le condizioni sono settate esplicitamente nei registri (istruzioni SET)
- E' presente un'unica modalità di indirizzamento in memoria (indiretto, mediante registro + offset)
- Le operazioni aritmetico/logiche sono eseguite solo tra registri (non tra registri e memoria)
- Esistono alcune istruzioni (MOVS2I e MOVI2S) per spostare dati tra registri GP e registri speciali e viceversa

# Registri del DLX (integer)



# DLX (integer): tipi di dato

Nel DLX (integer) sono disponibili tre tipi di dato:

**BYTE (8 bit)**



**HALF-WORD (16 bit)**



**WORD (32 bit)**



- I dati di dimensione inferiore a 32 bit (quindi a 8 o 16 bit) letti dalla memoria debbono essere estesi a 32 bit durante il caricamento nei registri (sempre a 32 bit)
- Questa operazione può essere eseguita mantenendo o meno il segno del dato letto dalla memoria

# Estensione del segno

In molti casi è necessario estendere la rappresentazione di un dato codificato con  $n$  bit, in un dato con una rappresentazione a  $m$  bit (con  $m > n$ ). Per esempio, volendo trasferire un byte ( $n=8$ ) dalla memoria in un registro a 32 bit ( $m=32$ ) è necessario conoscere la modalità con la quale è rappresentato il dato letto.

**Esempio:** 10110101 (n=8)

Assumendo il dato senza segno (*unsigned*) , l'estensione a 32 bit avviene aggiungendo 24 zeri:

Assumendo il dato con segno (signed), l'estensione avviene replicando 24 volte il bit di segno

# Il set di istruzioni (integer) del DLX

- Le principali istruzioni aritmetiche e logiche

- Istruzioni logiche anche con op. immediato: AND, ANDI, OR, ORI, XOR, XORI
- Istruzioni aritmetiche: ADD, ADDI, SUB, SUBI
- Istruzioni di shift (a destra anche aritmetico): SLL<sup>1</sup>, SRL, SRA<sup>2</sup>
- Istruzioni di SET CONDITION: Sx, con x = {EQ, NE, LT, GT, LE, GE}

- Le principali istruzioni di trasferimento dati

- Load byte signed e unsigned (LB, LBU), load halfword signed e unsigned (LH, LHU), load word (LW)
- Store byte, store halfword, store word: SB, SH, SW
- Copia un dato da un registro GP a un registro speciale e viceversa MOVS2I e MOVI2S

- Le principali istruzioni di trasferimento del controllo

- Istruzioni di salto condizionato (PC+4 relative): BNEZ, BEQZ
- Istruzioni di salto incondizionato J: assoluto (con reg.) e PC-relative
- Istruzioni di chiamata a procedura Jump and Link (JAL). L'indirizzo di ritorno viene automaticamente salvato in R31. JAL con registro e immediato (PC-relative)
- Istruzione di ritorno dalla procedura di servizio delle interruzioni: RFE

1) Shift logico a sinistra e shift aritmetico a sinistra coincidono (entrano 0 nei bit meno significativi). Per questa ragione NON esiste SLA. Fare attenzione con shift a sinistra, non preserva il segno e può generare overflow

2) Trascinando a destra di una posizione un registro e inserendo a sinistra sempre il bit del segno si mantiene il segno del dato mentre lo si divide successivamente per 2

# Elenco istruzioni del DLX (integer)

Data Transfer	Aritmetiche/logiche	Controllo
LW Ra, Imm <sub>16bit</sub> (Rb)	ADD Ra, Rb, Rc	Sx Ra, Rb, Rc
LB Ra, Imm <sub>16bit</sub> (Rb)	ADDI Ra, Rb, Imm <sub>16bit</sub>	SxI Ra, Rb, Imm <sub>16bit</sub>
LBU Ra, Imm <sub>16bit</sub> (Rb)	ADDU Ra, Rb, Rc	BEQZ Ra, Imm <sub>16bit</sub>
LH Ra, Imm <sub>16bit</sub> (Rb)	ADDUI Ra, Rb, Imm <sub>16bit</sub>	BNEZ Ra, Imm <sub>16bit</sub>
LHU Ra, Imm <sub>16bit</sub> (Rb)	SUB Ra, Rb, Rc	J Imm <sub>26bit</sub>
SW Ra, Imm <sub>16bit</sub> (Rb)	SUBI Ra, Rb, Imm <sub>16bit</sub>	JR Ra
SH Ra, Imm <sub>16bit</sub> (Rb)	SUBU Ra, Rb, Rc	JAL Imm <sub>26bit</sub>
SB Ra, Imm <sub>16bit</sub> (Rb)	SUBUI Ra, Rb, Imm <sub>16bit</sub>	JALR Ra
MOVS2I Ra, Rs*	SLL Ra, Rb, Rc	<b>x</b> può essere: <b>LT, GT, LE, GE, EQ, NE</b>
MOVI2S Rs*, Ra	SLLI Ra, Rb, Imm <sub>16bit</sub>	
Special register Rs* (IAR)	SRL Ra, Rb, Rc	
Per le istruzioni aritmetiche: l'immediato a 16 bit è esteso senza segno se di tipo U (unsigned) altrimenti con segno.	SRLI Ra, Rb, Imm <sub>16bit</sub>	
Per istruzioni logiche, sempre estensione senza segno.	SRA Ra, Rb, Rc	
	SRAI Ra, Rb, Imm <sub>16bit</sub>	
	OR Ra, Rb, Rc	
	ORI Ra, Rb, Imm <sub>16bit</sub>	
	XOR Ra, Rb, Rc	
	XORI Ra, Rb, Imm <sub>16bit</sub>	
	AND Ra, Rb, Rc	
	ANDI Ra, Rb, Imm <sub>16bit</sub>	
	LHI Ra, Imm <sub>16bit</sub>	

Ra  $\in \{R0^+, R1, \dots, R30, R31\}$

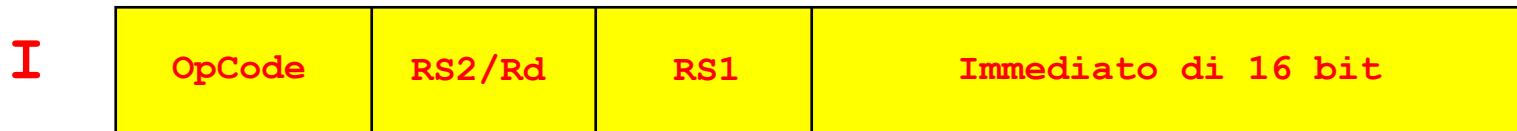
Rb  $\in \{R0, R1, \dots, R30, R31\}$

Rc  $\in \{R0, R1, \dots, R30, R31\}$

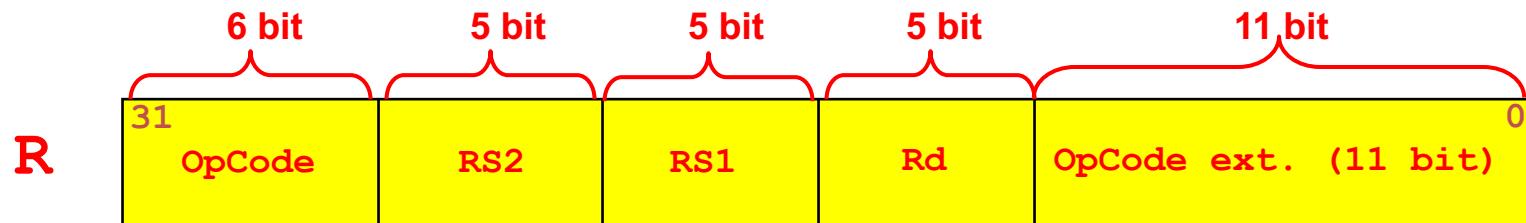
+Ra non può essere R0 come registro destinazione di istruzioni load, MOV2SI, aritmetico/logiche, LHI e SET

# DLX: formato delle istruzioni

- In alcune istruzioni di tipo I (LOAD e ALU), RS2 rappresenta il registro destinazione Rd
- Alcune istruzioni (e.g., J e JAL con registro) potrebbero essere codificate con più di un formato tra quelli disponibili



- Istruzioni di load, store, branch, J e JAL con registro, set condition SxI e ALU con operando immediato. L'immediato è a 16 bit
- Nelle operazioni load e ALU RS2/Rd è Rd. Nelle store RS2/Rd è RS2. In entrambi i casi RS1 per indirizzo sorgente (load o store) o registro sorgente (operazioni ALU con operando immediato)



- Istruzioni ALU del tipo  $Rd \leftarrow Rs1 \text{ op } Rs2$  oppure set condition Sx tra registri



- Salti incondizionato con e senza ritorno (J e JAL) con immediato

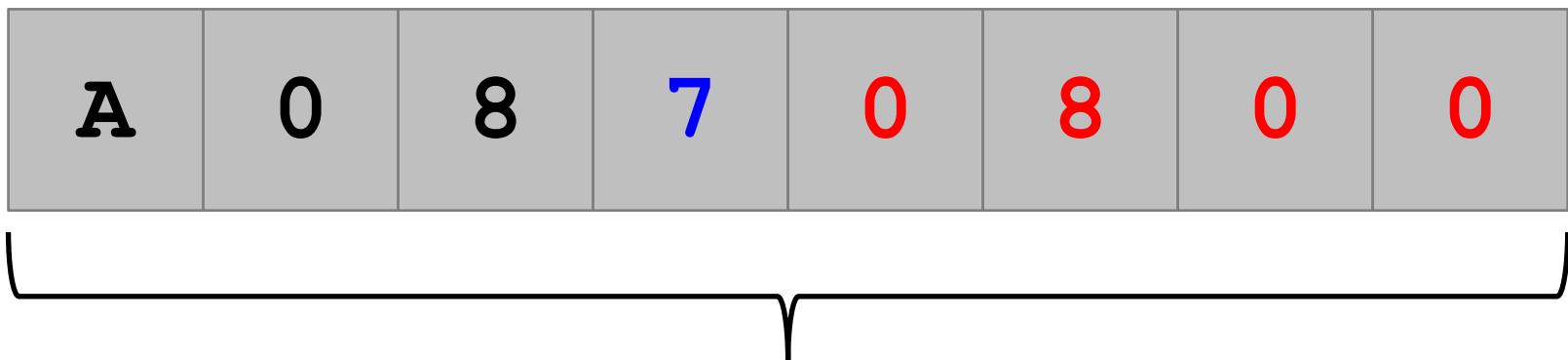
# Modalità di accesso alla memoria

- Ogni ISA dispone di istruzioni per accedere alla memoria in lettura e scrittura
- Normalmente è possibile stabilire la dimensione del dato che può essere trasferito (BYTE, HALF-WORD, WORD, etc)
- I due principali metodi di accesso alla memoria (indirizzamento) sono:
  - **Diretto** (indirizzo cablato nell'istruzione)
  - **Indiretto** (indirizzo modificabile a *run-time*)

# Indirizzamento diretto

- Con questa modalità l'istruzione contiene al suo interno un valore (cablato) che specifica l'indirizzo di accesso alla memoria

LB R7,0800h - "Leggi un BYTE (8 bit) all'indirizzo 0800h e memorizzala nel registro R7"

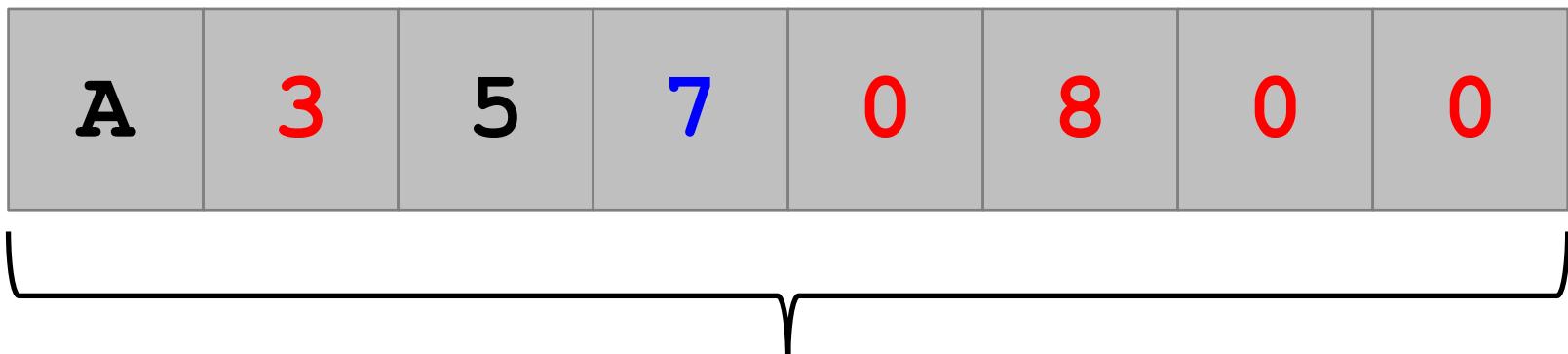


Ipotetica codifica dell'istruzione con 32 bit

# Indirizzamento indiretto

- Con questa modalità l'indirizzo di accesso alla memoria è ottenuto sommando un valore costante presente nell'istruzione con il contenuto di un registro
- **Indirizzo = costante + registro**
- Il registro è cablato nell'istruzione ma il suo contenuto può cambiare a tempo di esecuzione

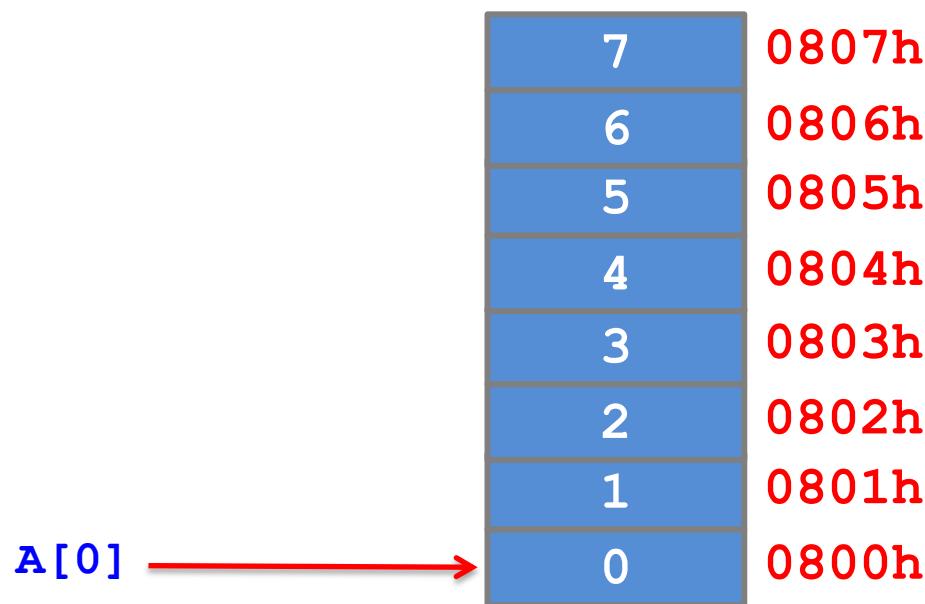
LB R7,0800(R3) - "Leggi un BYTE (8 bit) all'indirizzo R3 + 0800h e memorizzala nel registro R7"



Ipotetica codifica dell'istruzione con 32 bit

# Indirizzamento diretto vs indiretto 1/2

- La differenza tra le due modalità di indirizzamento è notevole
- Per rendervene conto, potete considerare un caso piuttosto comune: “*sommare gli elementi di un array A di 8 elementi memorizzato a partire dall’indirizzo 0800h*”



# Indirizzamento *diretto* vs *indiretto* 2/2

**Diretto (non usato) :**

```
XOR R8,R8,R8 ; R8=0  
  
LBU R7,0800h  
ADD R8,R8,R7 ; R8=R8+R7  
LBU R7,0801h  
ADD R8,R8,R7 ; R8=R8+R7  
LBU R7,0802h  
ADD R8,R8,R7 ; R8=R8+R7  
LBU R7,0803h  
ADD R8,R8,R7 ; R8=R8+R7  
LBU R7,0804h  
ADD R8,R8,R7 ; R8=R8+R7  
LBU R7,0805h  
ADD R8,R8,R7 ; R8=R8+R7  
LBU R7,0806h  
ADD R8,R8,R7 ; R8=R8+R7  
LBU R7,0807h  
ADD R8,R8,R7 ; R8=R8+R7
```

**Indiretto:**

```
XOR R8,R8,R8 ; R8=0  
ADDI R9,R8,8 ; R9=8  
  
LOOP: SUBI R9,R9,1 ; R9=R9-1  
      LBU R7,0800h(R9) ; legge BYTE  
                  ; a 0800+R9  
      ADD R8,R8,R7 ; R8=R8+R7  
      BNEZ R9,LOOP ; se R9!=0  
                  ; salta a LOOP
```

- Il registro R9 è utilizzato in ogni iterazione per cambiare l'indirizzo base (0800h)
- Pensate se l'array fosse di 1000000 elementi...

## DLX: modalità di accesso alla memoria

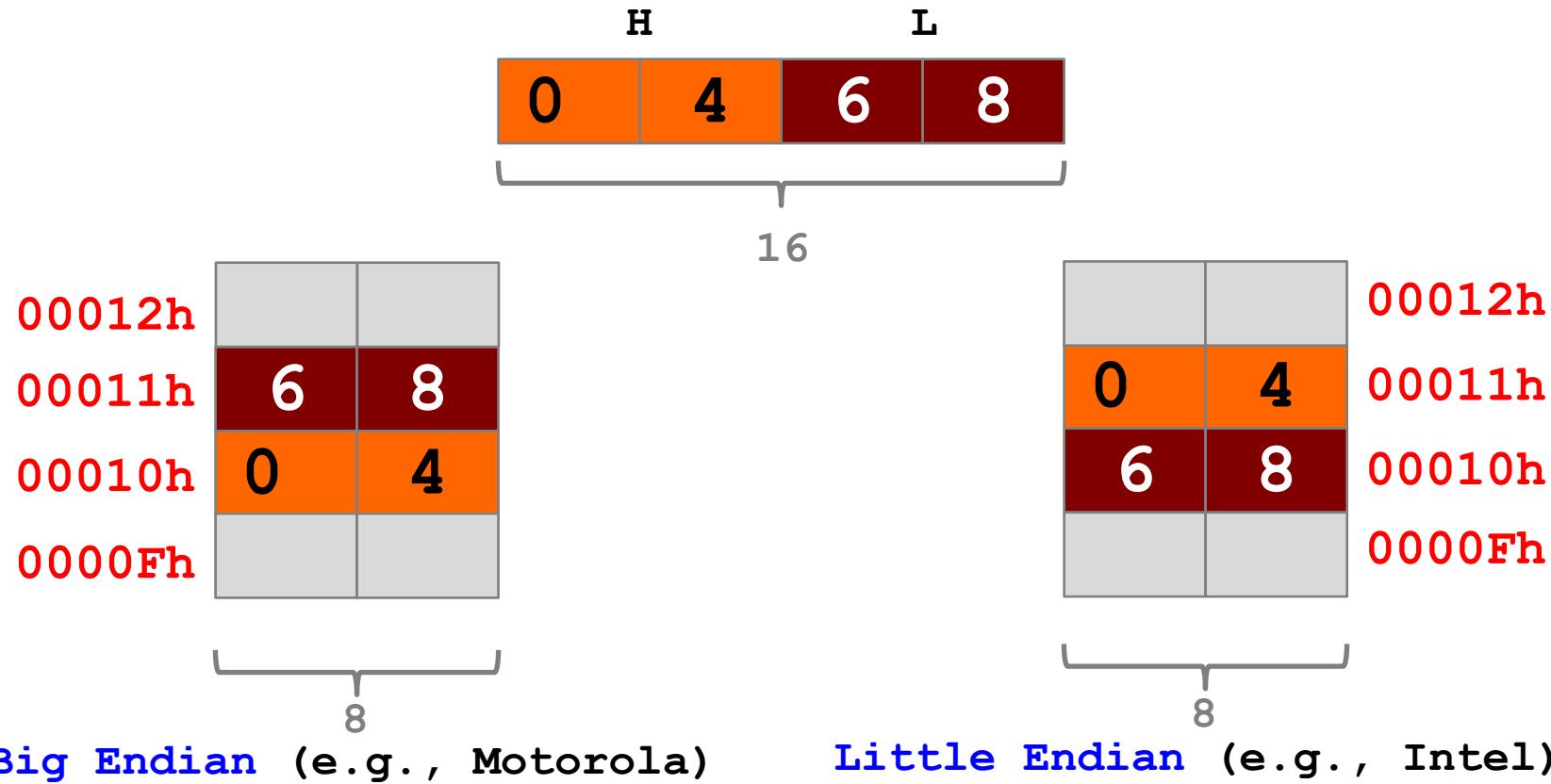
- Il DLX prevede un'unica modalità di indirizzamento: **indiretto**
- L'indirizzo (a 32 bit) è sempre ottenuto sommando un registro a 32 bit con un valore immediato a 16 bit esteso a 32 bit con segno.
- Esempio: LW R7, Imm<sub>16\_bit</sub>(R8)
- Carica in R7, la word all'indirizzo (a 32 bit) ottenuto sommando R8 il valore dell'immediato esteso a 32 bit con segno:

$$R7 \leftarrow_{32} M[R8 + Imm_{16\_bit}[15]^{16}\#Imm_{16\_bit}[15..0]]$$

- Nell'esercizio delle pagine precedenti abbiamo sottointeso, per semplicità, che l'indirizzo fosse a 16 bit. In realtà, nel DLX l'indirizzo è sempre a 32 bit (lo spazio di indirizzamento è 4 G)

# Come sono memorizzati i dati in memoria in un sistema con parallelismo > 8?

- Consideriamo un sistema con bus dati a 16 bit
- Come possiamo memorizzare il valore a 16 bit **0468h** a partire dall'indirizzo (che supponiamo a 20 bit) **00010h**?
- Esistono due convenzioni:



# Istruzioni Aritmetico Logiche (ALU)

- Istruzioni a 3 operandi:
  - 2 operandi “*sorgente*”
  - 1 operando “*destinazione*”.
- “*destinazione*”: sempre un registro (a 32 bit)
- “*sorgente*”: registro, registro
- “*sorgente*”: operando immediato (16 bit)

Esempi:

**ADD R1,R2,R3** ;  $R1 \leftarrow R2 + R3$  formato R

**ADDI R1,R2,3** ;  $R1 \leftarrow R2 + 3$  formato I  
; il valore (3) dell’immediato a 16 bit  
; è esteso a 32 bit

# Istruzioni di *Set Condition*

Queste istruzioni confrontano i due operandi *sorgente* e mettono a "1" oppure a "0" l'operando *destinazione* in funzione del risultato del confronto

- "SET EQUAL" (SEQ, =) : setta se uguale
- "SET NOT EQUAL" (SNE, !=) : setta se diverso
- "SET LESSER THAN" (SLT, <) : setta se <
- . . .

Gli operandi possono anche essere *unsigned*:

- "SET LESSER THAN UNSIGNED" (SLTU, <)

Esempi

**SLT R1 ,R2 ,R3** ; R1  $\leftarrow$  1 se R2<R3 altrimenti R1  $\leftarrow$  0  
; formato R

**SLTI R1 ,R2 ,3** ; R1  $\leftarrow$  1 se R2<3 altrimenti R1  $\leftarrow$  0  
; formato I

# Istruzioni per il *trasferimento dati*

- Sono istruzioni che accedono alla memoria (load e store):  
**LB, LBU, SB, LH, LHU, SH, LW, SW**
- L'indirizzo dell'operando in memoria è la somma del contenuto di un registro a 32 bit con un "offset" di 16 bit ( $\text{Imm}_{16\text{bit}}$ ) esteso con segno a 32 bit
- L'istruzione è codificata secondo il formato I

Esempi:

**LW R1,40(R3)** ;  $R1 \leftarrow_{32} M[40+R3]$

**LB R1,40(R3)** ;  $R1 \leftarrow_{32} (M[40+R3]_7)^{24} \# M[40+R3]$

**LBU R1,40(R3)** ;  $R1 \leftarrow_{32} (0)^{24} \# M[40+R3]$

# Istruzioni per il *trasferimento del controllo*: salti incondizionati (con e senza ritorno)

- “**JUMP**” : salto incondizionato
- “**JUMP AND LINK**” : salto incondizionato con ritorno

## Esempi

**J offset** ; PC = PC + 4 + (offset[25])<sup>6</sup> ## offset, tipo J  
**JR R3** ; PC = R3, tipo R

**JAL offset** ; R31 = PC+4  
; PC = PC + 4 + (offset[25])<sup>6</sup> ## offset, tipo J

**JALR R5** ; R31 = PC + 4, PC = R5

**JR R31** ; PC = R31  
; istruzione per tornare da una procedura

# Istruzioni per il *trasferimento del controllo*: salti condizionati (Branch)

**BCOND Rd, Imm<sub>16</sub>**

E' possibile verificare solo due condizioni (**COND**):

- **BEQZ "BEQUAL ZERO"** : salta se registro è 0
- **BNEQZ "BRANCH NOT EQUAL ZERO"** : salta se registro è ≠ 0

Esempi

```
BEQZ R4, Imm16 ; se R4==0 PC = PC + 4 + Imm16[15]16 ## Imm16
                    ; altrimenti PC = PC + 4
BNEZ R4, Imm16 ; se R4!=0 PC = PC + 4 + Imm16[15]16 ## Imm16
                    ; altrimenti PC = PC + 4
```

Con una istruzione di tipo **set** seguita da un'istruzione di **branch** si realizza la funzione di **compare and branch** (confronto e salto condizionato dal risultato del confronto) senza bisogno di flag dedicati

# Come generare valori a 32 bit

Nel DLX è presente una istruzione, LHI ("Load High Immediate") che consente di creare rapidamente valori a 32 bit (NON è una istruzione di accesso alla memoria!).

**LHI Rd, Imm<sub>16</sub>** ; Rd = Imm<sub>16</sub> ## 0000h

Inserisce in Rd il valore dell'immediato nei 16 bit più significativi e 0 nei rimanenti bit

Tipicamente, LHI è utilizzata per generare indirizzi a 32 bit partendo da immediati a 16 bit.

**Quali potrebbero essere delle alternative alla LHI?**

**Esempio**

**LHI R1, 8420** ; R1 = 8420 ## 0000h = 84200000h

# Esempio di codice assembler DLX 1

Quale valore assumono i registri R3 ed R4 al termine dell'esecuzione del codice seguente?

```
LHI      R1 ,0xE000
ADDUI   R2 ,R0 ,0x0081
SB       0x0000 (R1) ,R2
LBU      R3 ,0x0000 (R1)
LB       R4 ,0x0000 (R1)
```

R3 = ? , R4 = ?

# Esercizio

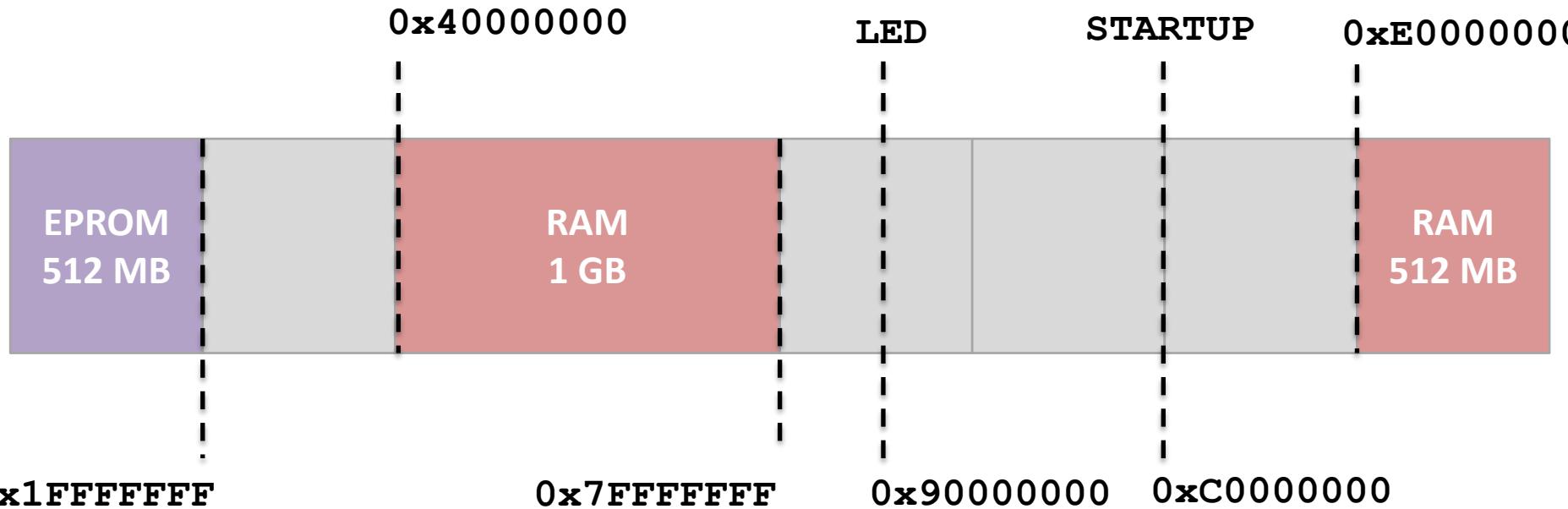
Scrivere il codice DLX che somma due word memorizzate a partire dalla metà dello spazio di indirizzamento (80000000h) .

```
LHI    R4,8000h      ; R4=8000##0000h
LW     R5,0(R4)       ; R5 <- M[R4+0]
LW     R6,4(R4)       ; R6 <- M[R4+4]
ADD   R7,R5,R6       ; R7 = R5 + R6
```

# Esercizio

Progettare un sistema basato sul processore DLX con **512 MB di EPROM nella parte bassa** dello spazio di indirizzamento, **1 GB di RAM a partire dall'indirizzo 0x40000000** e **512 MB di RAM nella parte finale** dello spazio di indirizzamento. Nel sistema, mediante opportune istruzioni software, è necessario poter:

- impostare al livello logico 0 o 1 un segnale denominato **STARTUP**, mappato a **0xC0000000** e **inizialmente al valore 1**
- **invertire lo stato di un LED, inizialmente spento** e mappato all'indirizzo **0x90000000**, prevedendo anche la possibilità di poterne leggere lo stato (ie, determinare se il LED è acceso o spento)



# Struttura di una soluzione

Cognome	Nome	Matricola	Data	Tipo esame (Calcolatori T o LA)
---------	------	-----------	------	---------------------------------

Risposta a eventuali domande specifiche indicate nel testo

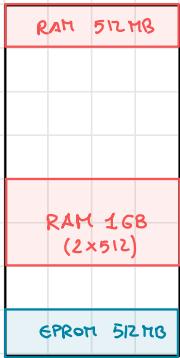
Indicazione dei dispositivi e delle memorie utilizzati con relativi indirizzi di mapping reali (inizio e fine in esadecimale)

Scrittura dei chip-select di ciascun dispositivo con decodifica semplificata

Progetto di eventuali reti logiche necessarie per risolvere il problema

Indicazione di come sono connessi tutti i dispositivi (incluse tutte le memorie) presenti nella souzione ai bus di sistema del DLX

Scrittura del codice in assembler DLX, necessario a risolvere il problema



$$\begin{aligned} CS\_EPROM\_512\_3 &= \overline{BA31} \cdot \overline{BA30} \cdot BE3 \\ CS\_EPROM\_512\_2 &= " \cdot " \cdot BE2 \\ CS\_EPROM\_512\_1 &= " \cdot " \cdot BE1 \\ CS\_EPROM\_512\_0 &= " \cdot " \cdot BE0 \end{aligned}$$

$$CS\_RAM\_1\_GB\_L\_3 = \overline{BA31} \ BA30 \ \overline{BA29} \ BE3$$

:

$$CS\_RAM\_1\_GB\_H\_3 = \overline{BA31} \ BA30 \ BA29 \ BE3$$

:

Assegniamo 1 GB a LED

$$CS\_READ\_LED = BA31 \ BA30 \cdot \overline{BA2} \cdot MEMRD$$

$$CS\_SWITCH\_LED = BA31 \cdot \overline{BA30} \cdot BA2 \rightarrow XXX+4$$

Assegno 512 MB a STARTUP

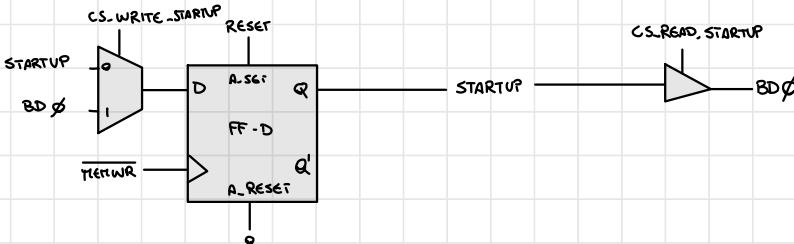
$$CS\_READ\_STARTUP = BA31 \ BA30 \ \overline{BA29} \cdot \overline{BA2} \cdot MEMRD$$

$$CS\_WRITE\_STARTUP = BA31 \ BA30 \ \overline{BA29} \cdot BA2 \leftarrow \text{non c'è MEMWR perciò va a sovrapposizione al fronte del clock}$$

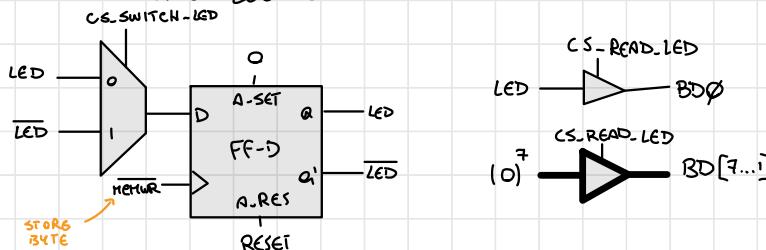
$$CS\_RAM\_512\_3 = BA31 \ BA30 \ BA29 \ BE3$$

:

All'arrivo del sistema  $STARTUP = 1$



All'arrivo del sistema  $LED = 0$



$$\begin{aligned} CS\_EPROM\_512\_3 &= \overline{BA31} \cdot \overline{BA30} \cdot BE3 \\ CS\_EPROM\_512\_2 &= " \cdot " \cdot BE2 \\ CS\_EPROM\_512\_1 &= " \cdot " \cdot BE1 \\ CS\_EPROM\_512\_0 &= " \cdot " \cdot BE0 \end{aligned}$$

$$CS\_RAM\_1\_GB\_L\_3 = \overline{BA31} \ BA30 \ \overline{BA29} \ BE3$$

:

$$CS\_RAM\_1\_GB\_H\_3 = \overline{BA31} \ BA30 \ BA29 \ BE3$$

:

Non esistono  
RAM da 1 GB  
ma 2 da 512 MB

AVERE POTUTO  
SCHEGLIERE  
UN ALTRO BIT  
INVECE DI  
BA2, MA È  
ARBITRARIO

Assegniamo 1 GB a LED

$$CS\_READ\_LED = BA31 \ BA30 \cdot \overline{BA2} \cdot MEMRD$$

$$CS\_SWITCH\_LED = BA31 \cdot \overline{BA30} \cdot BA2 \rightarrow XXX+4$$

Assegno 512 MB a STARTUP

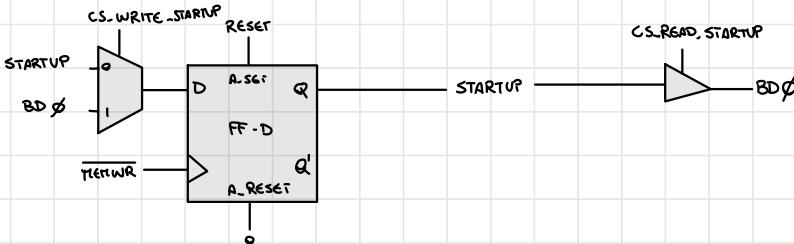
$$CS\_READ\_STARTUP = BA31 \ BA30 \ \overline{BA29} \cdot \overline{BA2} \cdot MEMRD$$

$$CS\_WRITE\_STARTUP = BA31 \ BA30 \ \overline{BA29} \cdot BA2 \leftarrow \text{non c'è MEMWR perciò va a sovrapposizione al fronte del clock}$$

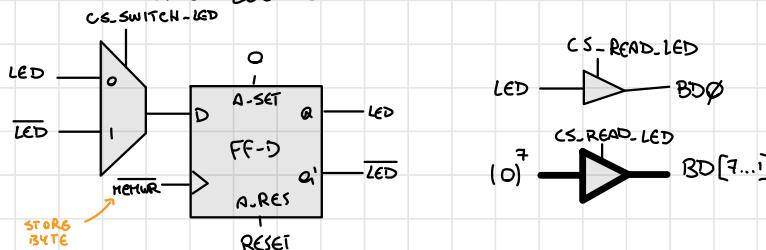
$$CS\_RAM\_512\_3 = BA31 \ BA30 \ BA29 \ BE3$$

:

All'arrivo del sistema  $STARTUP = 1$



All'arrivo del sistema  $LED = 0$



# Soluzione - chip select

CS\_EPROM\_512\_3 =  
CS\_EPROM\_512\_2 =  
CS\_EPROM\_512\_1 =  
CS\_EPROM\_512\_0 =

CS\_RAM\_1\_GB\_L\_3 =  
CS\_RAM\_1\_GB\_L\_2 =  
CS\_RAM\_1\_GB\_L\_1 =  
CS\_RAM\_1\_GB\_L\_0 =

CS\_RAM\_1\_GB\_H\_3 =  
CS\_RAM\_1\_GB\_H\_2 =  
CS\_RAM\_1\_GB\_H\_1 =  
CS\_RAM\_1\_GB\_H\_0 =

CS\_READ\_LED = (0x90000000)  
CS\_SWITCH\_LED = (0x90000004)

CS\_READ\_STARTUP = (0xC0000000)  
CS\_WRITE\_STARTUP = (0xC0000004)

CS\_RAM\_512\_3 =  
CS\_RAM\_512\_2 =  
CS\_RAM\_512\_1 =  
CS\_RAM\_512\_0 =

# **Soluzione – rete segnale STARTUP**

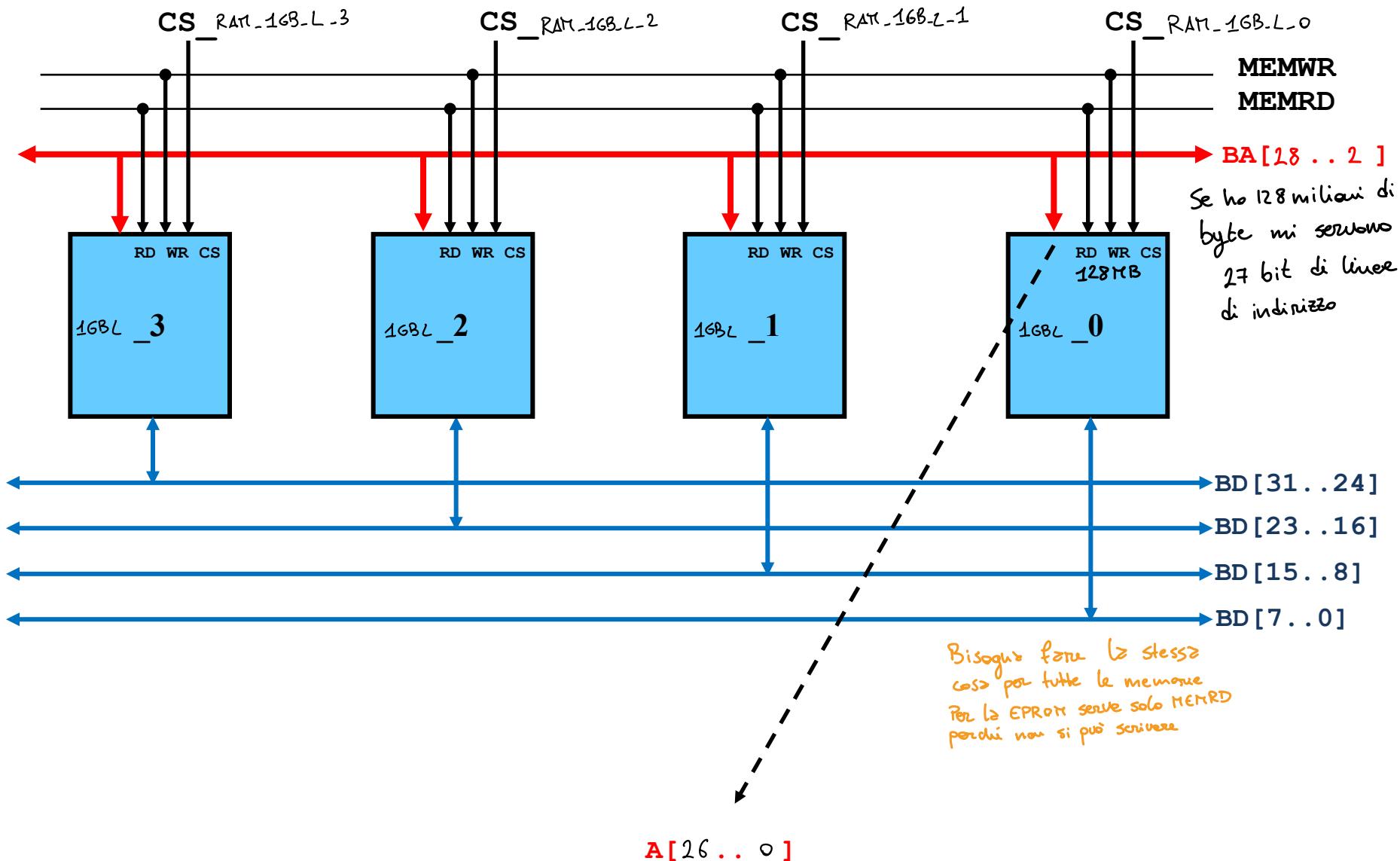
All'avvio del sistema STARTUP = 1

# **Soluzione – rete segnale LED**

All'avvio del sistema LED = 0

# Soluzione – connessione memorie

RAM 1GB-L



# Soluzione – codice assembler DLX 1/2

Lettura segnale STARTUP

CS-READ-STARTUP = 0xC0000000

LHI R10, 0xC0000000 ; R10 = 0xC0000000  
LBU R1, 0x000(R10) ; R1 ← M[R10]

Impostazione segnale STARTUP al valore logico 0

LHI R3, 0x000 ; R3 ← 0x000  
SB R0, 0x0004(R3); M[R3+4] ← R0

# Soluzione – codice assembler DLX 2/2

Lettura segnale LED

LHI R2, 0x9000 ; R2 = 0x90000000

LBW R3, 0x0000(R2)

Inversione valore del segnale LED

CS-SWITCH.LED = 0x90000004

LHI R2, 0x9000

SB R0, 0x0004(R2)

# **Simulatore DLX**

Nell'ambito di alcune testi di laurea è stato sviluppato un simulatore di istruzioni DLX per scopi didattici disponibile a questo indirizzo:

<http://dlx-simulator.disi.unibo.it/dlx>

Ancora in fase di sperimentazione ma potrebbe essere una valida alternativa al software indicato nelle pagine seguenti.

Per chi fosse interessato, sebbene sia ancora in forma molto preliminare, è possibile simulare anche istruzioni RISC-V

Sono gradite segnalazioni di bachi e suggerimenti per migliorare i simulatori in prossime tesi

**Tesi di laurea svolte in questo contesto:**

Federico Pomponii, "Sviluppo di un simulatore DLX per scopi didattici", AA 2019/20

Fabrizio Maccagnani, "Progetto di un simulatore di DLX per scopi didattici", AA 2018/19

Alessandro Foglia, "Progetto di un simulatore di RISC-V per scopi didattici", AA 2018/19

## Alcune note sul simulatore DLX attuale

- Negli immediati necessario il prefisso 0x (eg, 0x8000)
- Nelle store la destinazione a sinistra (eg, SW 0x800(R0),R18)
- Altro?

### Esempio: somma elementi di un array

```
init:    XOR R8,R8,R8          ; R8=0
         ADDI R9,R8,0x0008      ; R9=8
LOOP:    SUBI R9,R9,0x0001      ; R9=R9-1
         LBU R7,0x0800(R9)     ; legge un BYTE a 00000800+R9
         ADDU R8,R8,R7          ; R8=R8+R7
         BNEZ R9,LOOP           ; se R9!=0 salta a LOOP
```

0h	init:	XOR R8,R8,R8	; R8=0
4h		ADDI R9,R8,0x0008	; R9=8
8h	LOOP:	SUBI R9,R9,0x0001	; R9=R9-1
Ch		LBU R7,0x0800(R9)	; legge un BYTE a 00000800+R9
10h		ADDU R8,R8,R7	; R8=R8+R7
14h		BNEZ R9,LOOP	; se R9≠0 salta a LOOP

```

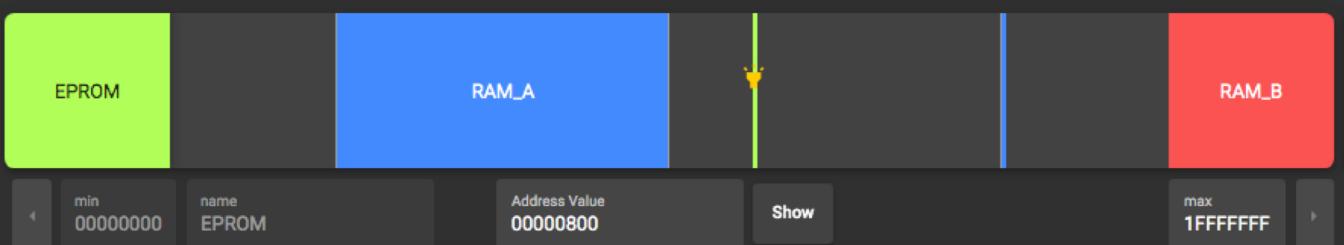
0 h init: XOR R8,R8,R8 ; R8=0
4 h ADDI R9,R8,0x0008 ; R9=8
8 h LOOP: SUBI R9,R9,0x0001 ; R9=R9-1
Ch LBU R7,0x0800(R9) ; legge un BYTE a 00000800+R9
10 h ADDU R8,R8,R7 ; R8=R8+R7
14 h BNEZ R9,LOOP ; se R9≠0 salta a LOOP
18 h
1Ch

```

GPR	PC
R0 0x00000000	0x000000024
R1 0x00000000	IAR
R2 0x00000000	0x00000000
R3 0x00000000	IEN
R4 0x00000000	0
R5 0x00000000	MAR
R6 0x00000000	0x00000800
R7 0x00000001	IR
R8 0x00000008	0x00000000
R9 0x00000000	MDR
R10 0x00000000	0x01010101
R11 0x00000000	
R12 0x00000000	
R13 0x00000000	
R14 0x00000000	
R15 0x00000000	
R16 0x00000000	
R17 0x00000000	
R18 0x00000000	
R19 0x00000000	
R20 0x00000000	
R21 0x00000000	
R22 0x00000000	
R23 0x00000000	
R24 0x00000000	
R25 0x00000000	
R26 0x00000000	
R27 0x00000000	
R28 0x00000000	
R29 0x00000000	
R30 0x00000000	
R31 0x00000000	

Formato valori  
Hexadecimal

Step Pause Interval (ms) 1000 Stop Start tag init Interrupt Save code Restore default code



# Esempio di codice assembler DLX 2

E' corretto il codice seguente?

LHI R1, 0x0000 ; R1 = 0x 0000 0000

ADDI R2, R0, 0x0081 ; R2 = 0x 0000 0081

SH 0x7FF1(R1), R2 ; Scrive 16 bit di R2 all'indirizzo R1 + 0x7FF1 M[R1+00007FF1]K<sub>16</sub>R2

LHU R3, 0x7FF1(R1); R3 = R2

LH R4, 0x7FF1(R1); R4 = R2

↓ Non è allineato!

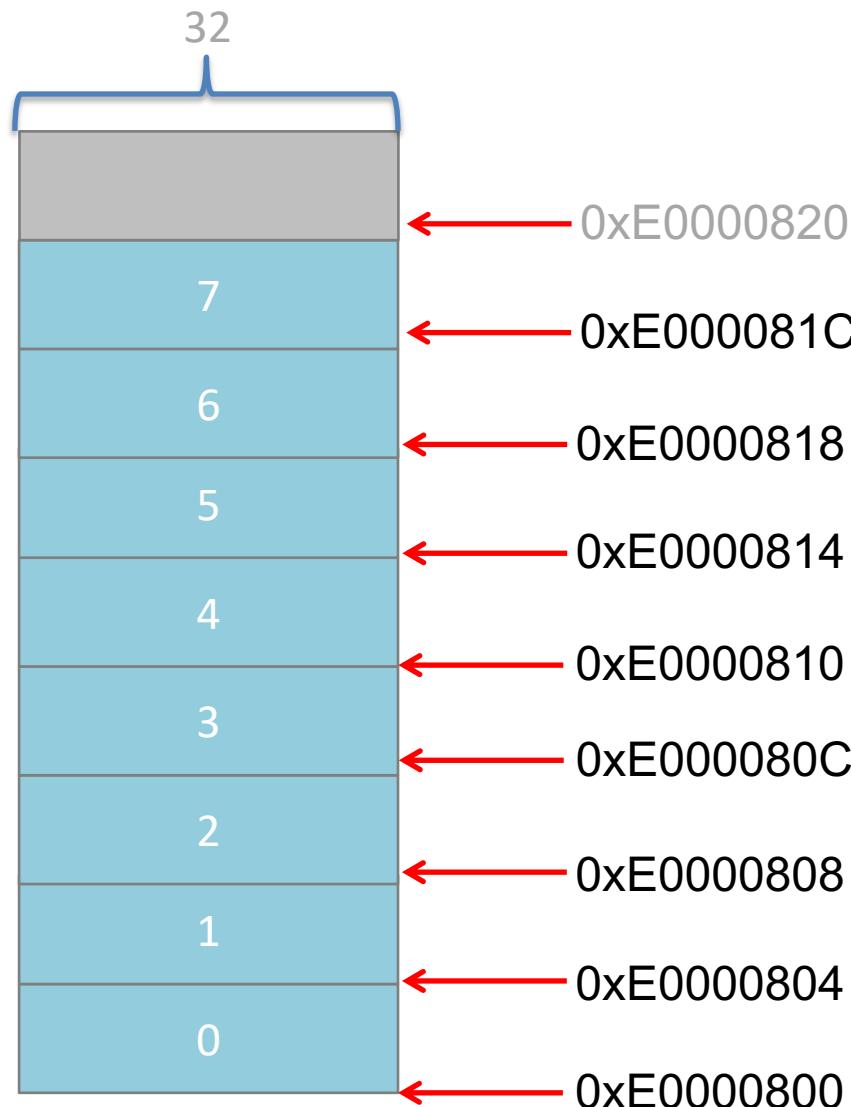
L'indirizzo non  
è allineato perché → Non si scrivono  
non è pari half-word a  
indirizzi dispari

Per le word  
→ multipli di 4

# Esempio di codice assembler DLX 3

Scrivere il codice assembler DLX per inserire in memoria, a partire dall'indirizzo **E0000800h** l'array di word indicato in figura.

```
LH1 R1, 0xE000  
LH1 R2, 0x0000  
LH1 R3, 0x0000  
ADD1 R3, R3, 0x0007  
SW R2, 0x0800(R1)  
  
Loop: ADD1 R1, R1, 0x0004  
      ADD1 R2, R2, 0x0001  
      SUB1 R3, R3, 0x0001  
      SW R2, 0x0800(R1)  
      BNEZ R3, Loop
```



```
ADD    R2,R0,R0      ; R2 usato come indice del ciclo  
LHI    R3,0xE000      ; R3 = E0000000
```

**Loop:**

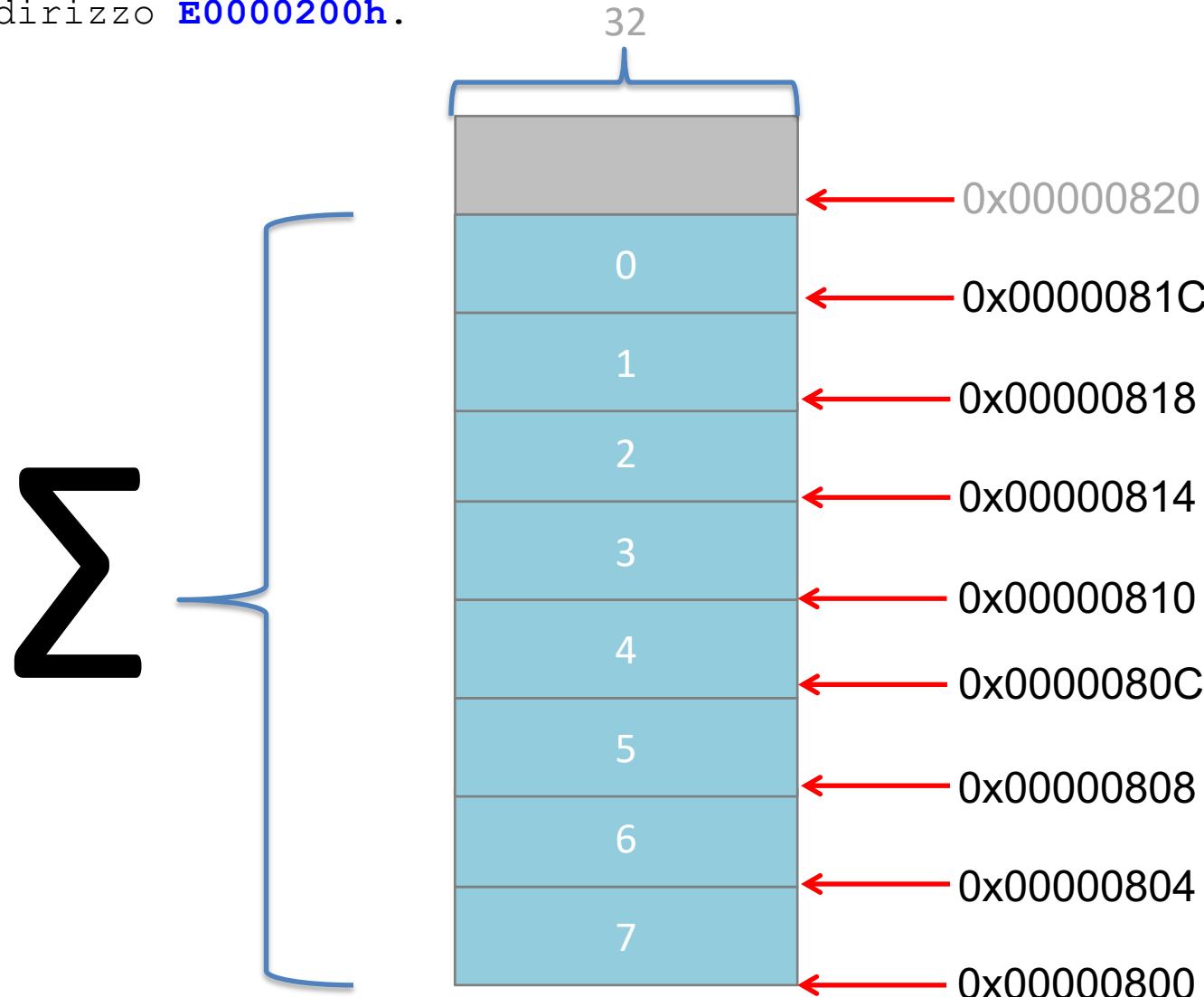
```
SW     R2,0800(R3)   ; scrive indice in memoria  
ADDI   R2,R2,1       ; incrementa indice del ciclo di 1  
ADDI   R3,R3,4       ; incrementa indice offset di 4  
SNEI   R4,R2,8       ; confronta indice R2 con 8  
BNEZ   R4,Loop      ; salta se R4 non è zero
```

Lo

- i) Quale valore è necessario sostituire a Loop in BNEZ?
- i) Si può fare meglio (ie, usare meno istruzioni)?

# Esempio di codice assembler DLX 4

Scrivere il codice assembler per il calcolo della somma dei primi 8 elementi di un vettore di WORD memorizzato a partire dall'indirizzo **00000800h**. Il risultato dell'elaborazione deve essere memorizzato all'indirizzo **E0000200h**.



	<b>ADD</b>	<b>R1 ,R0 ,R0</b>	; azzera R1, accumulatore
	<b>ADDI</b>	<b>R2 ,R0 ,20h</b>	; R2 = $32_{10}$
<b>Loop:</b>	<b>SUBI</b>	<b>R2 ,R2 ,4h</b>	; R2 = R2 - $4_{10}$
	<b>LW</b>	<b>R3 ,0800 (R2)</b>	; legge word in memoria
	<b>ADD</b>	<b>R1 ,R1 ,R3</b>	; aggiorna accumulatore R1
	<b>BNEZ</b>	<b>R2 ,Loop</b>	; salta se R2 non è zero
	<b>LHI</b>	<b>R7 ,0xE000</b>	; R7 = E0000000
	<b>SW</b>	<b>R1 ,0200h (R7)</b>	; memorizza accumulatore in E0000200h

# Esercizio

## Esercizio 1

In un sistema basato su un microprocessore DLX, con **512 MB di EPROM** mappata negli indirizzi bassi e **2 GB di RAM** mappata negli indirizzi alti, è necessario *contare modulo  $2^{16}$* , mediante una opportuna rete logica da progettare, **quanti accessi in scrittura** (senza distinzione tra *byte*, *halfword* e *word*) **sono stati eseguiti negli ultimi 512 MB** dello spazio di indirizzamento. La rete logica dovrà essere inizializzata automaticamente all'avvio e anche, in un qualsiasi istante, mediante un opportuno comando software. Inoltre, sempre mediante un opportuno comando software dovrà essere possibile leggere quanti accessi (modulo  $2^{16}$ ) all'area di memoria indicata in precedenza sono stati eseguiti dall'ultima volta che la rete logica è stata inizializzata.

Tutte le periferiche saranno utilizzate unicamente per le finalità indicate nel testo.

- **Descrivere sinteticamente la soluzione** che s'intende realizzare e indicare chiaramente quali sono i segnali di *chip-select* necessari
- Progettare il sistema **minimizzando le risorse necessarie e risolvendo eventuali criticità**
- Indicare le espressioni di decodifica e il range di indirizzi di tutte le periferiche, le memorie e i segnali
- Indicare esplicitamente le istruzioni che consentono di: inizializzare la rete logica e leggere il valore di conteggio
- Soluzioni **interamente software NON saranno considerate valide**

RAM 2GB

COUNTER  
X2^16

EPROM 64MB

CS\_EPROM\_0 = BA31 BA30 BE0

CS\_EPROM\_1 = " BE1

CS\_EPROM\_2 = " BE2

CS\_EPROM\_3 = " BE3

RAM\_2GB\_0 = BA31 BE0

RAM\_2GB\_1 = BA31 BE1

RAM\_2GB\_2 = BA31 BE2

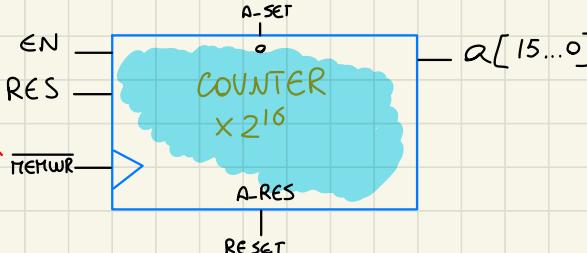
RAM\_2GB\_3 = BA31 BE3

CS\_EN\_COUNTER = BA31 BA30 BA29

CS\_RES\_COUNTER =  $\overline{BA31} \cdot BA30 \cdot BA2$

CS\_READ\_COUNTER =  $\overline{BA31} \cdot BA30 \cdot \overline{BA2} \cdot MEMRD$

NON SERVE  
MEMWR PERCHÉ  
È NEL CLOCK DEL  
COUNTER



RESET SINCRONO DEL COUNTER:

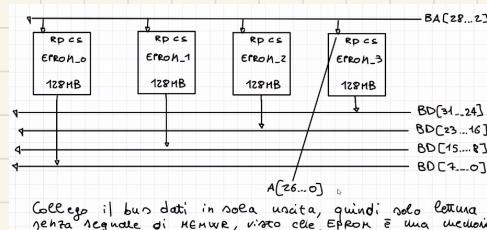
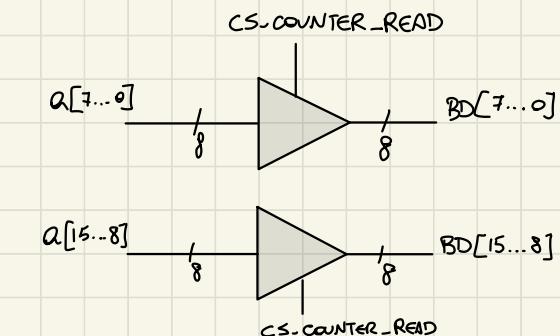
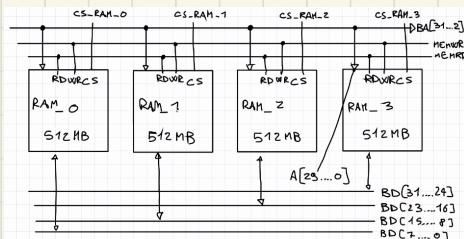
LHI R1 0x4000; R1 = 0x4000 0000

SW RO, 0x0004(R1); M[R1+4] ← RO

LETTURA DEL COUNTER:

LHI R2 0x4000; R2 = 0x4000 0000

LHU R3, 0x0000(R2); R3 ← M[R2]



Collego il bus dati in sola uscita, quindi solo lettura senza segnale di MEMWR, visto che EPROM è una memoria a sola lettura. BA1 e BA0 per codifica BE3, BE2, BE1, BE0

Si sono sicuri che il contatore  
sia stabile perché interviene  
MEMREAD e nel counter c'è  
MEMWRITE